Difference between HTTP1.1 vs HTTP2

HTTP/1.1 and HTTP/2 are two different versions of the Hypertext Transfer Protocol (HTTP) used for communication between clients (such as web browsers) and web servers. They have distinct differences in terms of performance, features, and how they handle requests and responses. Here's a comparison of HTTP/1.1 and HTTP/2:

Request and Response Multiplexing:

- **HTTP/1.1**: In HTTP/1.1, each request-response pair typically requires its own TCP connection. This can lead to inefficiencies, especially when multiple resources need to be fetched simultaneously, as browsers are limited in how many connections they can make to a single server (usually around 6-8 per domain).

- **HTTP/2**: HTTP/2 introduces multiplexing, which allows multiple requests and responses to be sent and received over a single TCP connection simultaneously. This reduces the need for multiple connections and significantly improves the efficiency of resource loading.

Header Compression:

- **HTTP/1.1**: HTTP/1.1 sends headers in plain text for each request and response. This can result in a significant amount of redundant data being transmitted, especially for large websites with many resources.
- **HTTP/2**: HTTP/2 uses header compression (HPACK) to reduce the overhead of sending headers, resulting in faster and more efficient data transfer.

Server Push:

- **HTTP/1.1**: Server push is not supported in HTTP/1.1. Clients must explicitly request each resource they need.
- **HTTP/2**: HTTP/2 supports server push, where the server can initiate the sending of resources to the client before the client requests them. This reduces latency and can lead to faster page loading times.

Prioritization:

- **HTTP/1.1**: Requests are processed in the order they are received, with no built-in mechanism for specifying the importance of one request over another.
- **HTTP/2**: HTTP/2 allows for request prioritization, so more critical resources can be loaded faster. This is especially useful for optimizing the loading of web pages.

Binary Protocol:

- **HTTP/1.1**: It uses a text-based protocol, which is human-readable but less efficient for machines to parse.
- **HTTP/2**: It uses a binary protocol, which is more efficient for machines to process. While it's not human-readable, it's faster to parse.

Backward Compatibility:

- **HTTP/1.1**: HTTP/1.1 is well-established and widely supported by both servers and clients. It is designed to be backward-compatible with HTTP/1.0, allowing gradual adoption of new features.
- **HTTP/2**: HTTP/2 is designed to be compatible with HTTP/1.1 at a high level but requires server and client support for its advanced features.

Objects and its internal representation in Javascript

In JavaScript, objects are a complex data type used to store a collection of key-value pairs. They are often referred to as unordered collections of properties. Properties can be thought of as variables that are part of the object. These properties can contain various data types, including other objects, functions, and primitive values like numbers and strings.

1. Properties:

Properties in JavaScript objects consist of two parts: a key (also known as a property name) and a value. The key is always a string (or a symbol, in the case of ES6 symbols), and the value can be any data type, including other objects.

2. Object Prototype:

Each JavaScript object has a prototype, which is an object from which it inherits properties and methods. You can access an object's prototype using the `Object.getPrototypeOf(obj)` method.

3. Methods:

Objects can contain methods, which are functions stored as property values. These methods can perform actions or calculations related to the object's data.

4. Object Creation:

There are multiple ways to create objects in JavaScript. The most common methods are object literals, constructor functions, and classes (introduced in ES6).

Object Properties and Methods

JavaScript provides several built-in methods and properties for working with objects. Here are a few commonly used ones:

1. `Object.keys(obj)`**:**

Returns an array of a given object's own enumerable property names

2. `Object.values(obj)`**:**

Returns an array of a given object's own enumerable property values.

3. `Object.entries(obj)`**:**

Returns an array of a given object's own enumerable property `[key, value]` pairs.

4. `obj.hasOwnProperty(key)`**:**

Checks if an object has a specific property as its own property (not inherited).